Factor cookbook

The Factor cookbook is a high-level overview of the most important concepts required to program in Factor.

Basic syntax cookbook

The following is a simple snippet of Factor code:

```
10 sq 5 - .
95
```

You can click on it to evaluate it in the listener, and it will print the same output value as indicated above.

Factor has a very simple syntax. Your program consists of *words* and *literals*. In the above snippet, the words are **sq**, **-** and **.**. The two integers 10 and 5 are literals.

Factor evaluates code left to right, and stores intermediate values on a *stack*. If you think of the stack as a pile of papers, then *pushing* a value on the stack corresponds to placing a piece of paper at the top of the pile, while *popping* a value corresponds to removing the topmost piece.

All words have a *stack effect declaration*, for example `( x y -- z )` denotes that a word takes two inputs, with `y` at the top of the stack, and returns one output. Stack effect declarations can be viewed by browsing source code, or using tools such as **see**; they are also checked by the compiler. See **Stack effect declarations**.

Coming back to the example in the beginning of this article, the following series of steps occurs as the code is evaluated:

Action	Stack contents
10 is pushed on the stack.	`10`
The **sq** word is executed. It pops one input from the stack - the integer 10 - and squares it, pushing the result.	`100`
5 is pushed on the stack.	`100 5`
The **-** word is executed. It pops two inputs from the stack - the integers 100 and 5 - and subtracts 5 from 100, pushing the result.	`95`
The **.** word is executed. It pops one input from the stack - the integer 95 - and prints it in the listener's output area.	

Factor supports many other data types:

```
10.5
"character strings"
{ 1 2 3 }
! by the way, this is a comment
#! and so is this
```

References

Factor's syntax can be extended, the parser can be called reflectively, and the **.** word is in fact a general facility for turning almost any object into a form which can be parsed back in again. If this interests you, consult the following sections:

- **Syntax**
- **The parser**
- **The prettyprinter**

Shuffle word and definition cookbook

The **dup** word makes a copy of the value at the top of the stack:

```
5 dup * .
25
```

The **sq** word is actually defined as follows:

```
: sq ( x -- y ) dup * ;
```

(You could have looked this up yourself by clicking on the **sq** word itself.)

Note the key elements in a word definition: The colon **:** denotes the start of a word definition. The name of the new word and a stack effect declaration must immediately follow. The word definition then continues on until the **;** token signifies the end of the definition. This type of word definition is called a *compound definition.*

Factor is all about code reuse through short and logical colon definitions. Breaking up a problem into small pieces which are easy to test is called *factoring.*

Another example of a colon definition:

```
: neg ( x -- -x ) 0 swap - ;
```

Here the **swap** shuffle word is used to interchange the top two stack elements. Note the difference that **swap** makes in the following two snippets:

```
5 0 -       ! Computes 5-0
5 0 swap -  ! Computes 0-5
```

Also, in the above example a stack effect declaration is written between (and) with a mnemonic description of what the word does to the stack. See **Stack effect declarations** for details.

For the curious...

This syntax will be familiar to anybody who has used Forth before. However, unlike Forth, some additional static checks are performed. See **Definition sanity checking** and **Stack effect checking**.

References

A whole slew of shuffle words can be used to rearrange the stack. There are forms of word definition other than colon definition, words can be defined entirely at runtime, and word definitions can be *annotated* with tracing calls and breakpoints without modifying the source code.

Shuffle words

Words

Generic words and methods

Developer tools

Control flow cookbook

A *quotation* is an object containing code which can be evaluated.

```
2 2 + .     ! Prints 4
[ 2 2 + . ] ! Pushes a quotation
```

The quotation pushed by the second example will print 4 when called by **call**.

Quotations are used to implement control flow. For example, conditional execution is done with **if**:

```
: sign-test ( n -- )
    dup 0 < [
        drop "negative"
    ] [
        zero? [ "zero" ] [ "positive" ] if
    ] if print ;
```

The **if** word takes a boolean, a true quotation, and a false quotation, and executes one of the two quotations depending on the value of the boolean. In Factor, any object not equal to the special value **f** is considered true, while **f** is false.

Another useful form of control flow is iteration. You can do something several times:

```
10 [ "Factor rocks!" print ] times
```

Now we can look at a new data type, the array:

```
{ 1 2 3 }
```

An array differs from a quotation in that it cannot be evaluated; it simply stores data.

You can perform an operation on each element of an array:

```
{ 1 2 3 } [ "The number is " write . ] each
The number is 1
The number is 2
The number is 3
```

You can transform each element, collecting the results in a new array:

```
{ 5 12 0 -12 -5 } [ sq ] map .
{ 25 144 0 144 25 }
```

You can create a new array, only containing elements which satisfy some condition:

```
: negative? ( n -- ? ) 0 < ;
{ -12 10 16 0 -1 -3 -9 } [ negative? ] filter .
{ -12 -1 -3 -9 }
```

References

Since quotations are objects, they can be constructed and taken apart at will. You can write code that writes code. Arrays are just one of the various types of sequences, and the sequence operations such as **each** and **map** operate on all types of sequences. There are many more sequence iteration operations than the ones above, too.

- **Combinators**
- **Sequence operations**

Dynamic variables cookbook

A symbol is a word which pushes itself on the stack when executed. Try it:

```
SYMBOL: foo
foo .
```

```
foo
```

Before using a variable, you must define a symbol for it:

```
SYMBOL: name
```

Symbols can be passed to the **get** and **set** words to read and write variable values:

```
"Slava" name set
name get print
```

```
Slava
```

If you set variables inside a **with-scope**, their values will be lost after leaving the scope:

```
: print-name ( -- ) name get print ;
"Slava" name set
[
    "Diana" name set
    "There, the name is " write  print-name
] with-scope
"Here, the name is " write  print-name
```

```
There, the name is Diana
Here, the name is Slava
```

References

There is a lot more to be said about dynamically-scoped variables and namespaces.

- **Dynamic variables**

Vocabularies cookbook

Rather than being in one flat list, words belong to vocabularies; every word is contained in exactly one. When parsing a word name, the parser searches through vocabularies. When working at the listener, a useful set of vocabularies is already available. In a source file, all used vocabularies must be imported.

For example, a source file containing the following code will print a parse error if you try loading it:

```
"Hello world" print
```

The **print** word is contained inside the **io** vocabulary, which is available in the listener but must be explicitly added to the search path in source files:

```
USE: io
"Hello world" print
```

Typically a source file will refer to words in multiple vocabularies, and they can all be added to the search path in one go:

```
USING: arrays kernel math ;
```

New words go into the **scratchpad** vocabulary by default. You can change this with **IN:**:

```
IN: time-machine
: time-travel ( when what -- ) frob fizz flap ;
```

Note that words must be defined before being referenced. The following is generally invalid:

```
: frob ( what -- ) accelerate particles ;
: accelerate ( -- ) accelerator on ;
: particles ( what -- ) [ (particles) ] each ;
```

You would have to place the first definition after the two others for the parser to accept the file. If you have a set of mutually recursive words, you can use **DEFER:**.

References

Parse-time word lookup

Words

The parser

Application cookbook

Vocabularies can define a main entry point:

```
IN: game-of-life
...
: play-life ( -- ) ... ;

MAIN: play-life
```

See **MAIN:** for details. The **run** word loads a vocabulary if necessary, and calls its main entry point; try the following, it's fun:

```
"tetris" run
```

Factor can deploy stand-alone executables; they do not have any external dependencies and consist entirely of compiled native machine code:

```
"tetris" deploy-tool
```

References

- **Vocabulary loader**
- **Application deployment**
- **Application deployment UI tool**
- **Scripting cookbook**

Scripting cookbook

Factor can be used for command-line scripting on Unix-like systems.

To run a script, simply pass it as an argument to the Factor executable:
```
./factor cleanup.factor
```

The script may access command line arguments by inspecting the value of the **command-line** variable. It can also get its own path from the **script** variable.

Example: ls

Here is an example implementing a simplified version of the Unix `ls` command in Factor:
```
USING: command-line namespaces io io.files
io.pathnames tools.files sequences kernel ;

command-line get [
    current-directory get directory.
] [
    dup length 1 = [ first directory. ] [
        [ [ nl write ":" print ] [ directory. ] bi ] each
    ] if
] if-empty
```

You can put it in a file named `ls.factor`, and then run it, to list the `/usr/bin` directory for example:
```
./factor ls.factor /usr/bin
```

Example: grep

The following is a more complicated example, implementing something like the Unix `grep` command:
```
USING: kernel fry io io.files io.encodings.ascii sequences
regexp command-line namespaces ;
IN: grep

: grep-lines ( pattern -- )
    '[ dup _ matches? [ print ] [ drop ] if ] each-line ;

: grep-file ( pattern filename -- )
    ascii [ grep-lines ] with-file-reader ;

: grep-usage ( -- )
    "Usage: factor grep.factor <pattern> [<file>...]" print ;

command-line get [
    grep-usage
] [
    unclip <regexp> swap [
        grep-lines
    ] [
        [ grep-file ] with each
    ] if-empty
] if-empty
```

You can run it like so,
```
./factor grep.factor '.*hello.*' myfile.txt
```

You'll notice this script takes a while to start. This is because it is loading and compiling the **regexp** vocabulary every time. To speed up startup, load the vocabulary into your image, and save the image:

```
USE: regexp
save
```

Now, the `grep.factor` script will start up much faster. See **Images** for details.

Executable scripts

It is also possible to make executable scripts. A Factor file can begin with a comment like the following:

```
#! /usr/bin/env factor
```

If the text file is made executable, then it can be run, assuming the `factor` binary is in your `$PATH`.

The space between `#!` and `/usr/bin/env` is necessary, since **#!** is a parsing word, and a syntax error would otherwise result.

References

- **Command line arguments**
- **Application cookbook**
- **Images**

Factor philosophy

Learning a stack language is like learning to ride a bicycle: it takes a bit of practice and you might graze your knees a couple of times, but once you get the hang of it, it becomes second nature.

The most common difficulty encountered by beginners is trouble reading and writing code as a result of trying to place too many values on the stack at a time.

Keep the following guidelines in mind to avoid losing your sense of balance:

Simplify, simplify, simplify. Break your program up into small words which operate on a few values at a time. Most word definitions should fit on a single line; very rarely should they exceed two or three lines.

In addition to keeping your words short, keep them meaningful. Give them good names, and make sure each word only does one thing. Try documenting your words; if the documentation for a word is unclear or complex, chances are the word definition is too. Don't be afraid to refactor your code.

If your code looks repetitive, factor it some more.

If after factoring, your code still looks repetitive, introduce combinators.

If after introducing combinators, your code still looks repetitive, look into using meta-programming techniques.

Try to place items on the stack in the order in which they are needed. If everything is in the correct order, no shuffling needs to be performed.

If you find yourself writing a stack comment in the middle of a word, break the word up.

Use **Cleave combinators** and **Spread combinators** instead of **Shuffle words** to give your code more structure.

Not everything has to go on the stack. The **namespaces** vocabulary provides dynamically-scoped variables, and the **locals** vocabulary provides lexically-scoped variables. Learn both and use them where they make sense, but keep in mind that overuse of variables makes code harder to factor.

Every time you define a word which simply manipulates sequences, hashtables or objects in an abstract way which is not related to your program domain, check the library to see if you can reuse an existing definition.

Write unit tests. Factor provides good support for unit testing; see **Unit testing**. Once your program has a good test suite you can refactor with confidence and catch regressions early.

Don't write Factor as if it were C. Imperative programming and indexed loops are almost always not the most idiomatic solution.

Use sequences, assocs and objects to group related data. Object allocation is very cheap. Don't be afraid to create tuples, pairs and triples. Don't be afraid of operations which allocate new objects either, such as **append**.

If you find yourself writing a loop with a sequence and an index, there's almost always a better way. Learn the **Sequence combinators** by heart.

If you find yourself writing a heavily nested loop which performs several steps on each iteration, there is almost always a better way. Break the problem down into a series of passes over the data instead, gradually transforming it into the desired result with a series of simple loops. Factor the loops out and reuse them. If you're working on anything math-related, learn **Vector operations** by heart.

If you find yourself wishing you could iterate over the datastack, or capture the contents of the datastack into a sequence, or push each element of a sequence onto the datastack, there is almost always a better way. Use **Sequence operations** instead.

Don't use meta-programming if there's a simpler way.

Don't worry about efficiency unless your program is too slow. Don't prefer complex code to simple code just because you feel it will be more efficient. The Factor compiler is designed to make idiomatic code run fast.

None of the above are hard-and-fast rules: there are exceptions to all of them. But one rule unconditionally holds: *there is always a simpler way*.

Factor tries to implement as much of itself as possible, because this improves simplicity and performance. One consequence is that Factor exposes its internals for extension and study. You even have the option of using low-level features not usually found in high-level languages, such as manual memory management, pointer arithmetic, and inline assembly code.

Unsafe features are tucked away so that you will not invoke them by accident, or have to use them to solve conventional programming problems. However when the need arises, unsafe features are invaluable, for example you might have to do some pointer arithmetic when interfacing directly with C libraries.

Pitfalls to avoid

Factor is a very clean and consistent language. However, it has some limitations and leaky abstractions you should keep in mind, as well as behaviors which differ from other languages you may be used to.

Factor only makes use of one native thread, and Factor threads are scheduled co-operatively. C library calls block the entire VM.

Factor does not hide anything from the programmer, all internals are exposed. It is your responsibility to avoid writing fragile code which depends too much on implementation detail.

If a literal object appears in a word definition, the object itself is pushed on the stack when the word executes, not a copy. If you intend to mutate this object, you must **clone** it first. See **Literals**.

Also, **dup** and related shuffle words don't copy entire objects or arrays; they only duplicate the reference to them. If you want to guard an object against mutation, use **clone**.

For a discussion of potential issues surrounding the **f** object, see **Booleans**.

Factor's object system is quite flexible. Careless usage of union, mixin and predicate classes can lead to similar problems to those caused by "multiple inheritance" in other languages. In particular, it is possible to have two classes such that they have a non-empty intersection and yet neither is a subclass of the other. If a generic word defines methods on two such classes, various disambiguation rules are applied to ensure method dispatch remains deterministic, however they may not be what you expect. See **Method precedence** for details.

If **run-file** throws a stack depth assertion, it means that the top-level form in the file left behind values on the stack. The stack depth is compared before and after loading a source file, since this type of situation is almost always an error. If you have a legitimate need to load a source file which returns data in some manner, define a word in the source file which produces this data on the stack and call the word after loading the file.

Next steps

Once you have read through **Your first program** and **Factor cookbook**, the best way to keep learning Factor is to start looking at some simple example programs. Here are a few particularly nice vocabularies which should keep you busy for a little while:

- **base64**
- **roman**
- **rot13**
- **smtp**
- **time-server**
- **tools.hexdump**
- **webapps.counter**

If you see code in there that you do not understand, use **see** and **help** to explore.